Задание:

1. Необходимо написать функцию nash_equilibrium(a), которая принимает матрицу выигрыша и возвращает значение игры и оптимальные стратегии первого и второго игроков.
2. Проиллюстрировать работу нашего кода путем решения нескольких игр и визуализации спектров оптимальных стратегий игроков в Jupyter. В частности, нужно привести игры, в которых:
    - спектр оптимальной стратегии состоит из одной точки (т.е. существует равновесие Нэша в чистых стратегиях),
    - спектр оптимальной стратегии неполон (т.е. некоторые чистые стратегии не используются),
    - спектр оптимальной стратегии полон.
3. Оформить решение в вид пакета
4. Написать unit-тесты для функции nash_equilibrium

Решение:

Что такое антагонистическая матричная игра?

Рассмотрим конфликт двух участников с противоположными интересами, математической моделью которого является игра с нулевой суммой.
Участники игры – лица, принимающие решения, – называются *игроками*.
Стратегия игрока – это осознанный выбор одного из множества возможных вариантов его действий.
Будем рассматривать конечные игры, в которых множества стратегий игроков конечны; стратегии первого игрока пронумеруем числами от 1 до m, а стратегии второго игрока — числами от 1 до n. Если первый игрок выбрал свою i-ю стратегию, а второй игрок – свою j-ю стратегию, то результатом такого совместного выбора будет платеж aij второго игрока первому (это не обязательно денежная сумма, а любая оценка полезности результата выбора игроками своих стратегий i и j).
Таким образом, конечная игра с нулевой суммой однозначно определяется матрицей (aij), i = 1,…, m, j = 1, …, n, которая называется матрицей выигрышей. Строки этой матрицы соответствуют стратегиям первого игрока, а столбцы – стратегиям второго игрока.

Партия игры происходит следующим образом: Пусть, например, первый игрок назвал номер i, а второй – j. Тогда второй игрок платит первому сумму aij. На этом партия игры заканчивается. Если aij > 0, то это означает, что при выборе первым игроком i-й стратегии, а вторым j-й выигрывает первый игрок; если же aij < 0, то это значит, что при данном выборе стратегий в выигрыше оказывается второй игрок. Цель каждого игрока – выиграть как можно большую сумму в результате большого числа партий.

Смысл названий «*конфликт с противоположными интересами*» и «*игра с*

нулевой суммой» состоит в том, что выигрыш каждого из игроков противоположен выигрышу противника, или, иначе, что сумма выигрышей игроков равна нулю.
Стратегия называется *чистой*, если выбор игрока неизменен от партии к партии. У первого игрока, очевидно, есть m чистых стратегий, а у второго – n. Рассмотрим описанную конфликтную ситуацию с точки зрения первого игрока.

Если мы выбираем свою i-ю стратегию (i-ю строку матрицы A), то второй игрок, будучи разумным, выберет такую стратегию j, которая обеспечит ему наибольший выигрыш (а нам, соответственно, наименьший), т. е. он выберет такой столбец j матрицы A, в котором платеж aij минимален. Переберем все наши стратегии i = 1, 2, …, m и выберем ту из них, при которой второй игрок, действуя максимально разумно, заплатит нам наибольшую сумму. Величина

$$\alpha = \max_{i = 1,\dots,m} \min_{j=1,\dots,n}(aij)$$

называется *нижней ценой игры*, а соответствующая ей стратегия первого игрока – *максиминной*. Аналогичные рассуждения можно провести точки зрения второго игрока: *верхняя цена игры:*

$$\beta = \min_{i = 1,\dots,m} \max_{j=1,\dots,n}(aij)$$

и соответствующая ей стартегия - *минимаксная* стратегия второго игрока. *Нижняя цена игры α* представляет собой максимальный гарантированный выигрыш первого игрока , а *верхняя цена* – величину, противоположную минимальному гарантированному проигрышу второго игрока. Если α = β, то говорят, что игра имеет *седловую точку* в чистых стратегиях, общее значение α и β называется при этом *ценой игры* и обозначается V = α = β. При этом стратегии игроков, соответствующие седловой точке, называются *оптимальными чистыми стратегиями*.

Смешанной стратегией первого игрока называется вектор $p = (p_1, \dots, p_m)$, где все $p_i >= 0$ (i=1, 2 ,…, m), а $\sum p_i = 1$. При этом p_i – вероятность, с которой первый игрок выбирает свою i-ю стратегию. Аналогично определяется смешанная стратегия $q = (q_1, \dots, q_n)$ второго игрока. Чистая стратегия также подпадает под определение смешанной – в этом случае все вероятности равны нулю, кроме одной, равной единице. Если игроки играют со своими смешанными стратегиями p и q соответственно, то математическое ожидание выигрыша первого игрока равно

$$M(p, q) = \sum\sum a_{ij} p_i q_i$$

(и совпадает с математическим ожиданием проигрыша второго игрока). Стратегии p* и q* называются *оптимальными смешанными стратегиями* соответственно первого и второго игрока, если

$$M(p, q^*) <= M(p^*, q^*) <= M(p^*, q).$$

Если у обоих игроков есть оптимальные смешанные стратегии, то пара (p*, q*) называется *решением игры* (или седловой точкой в смешанных стратегиях), а число V = M(p*, q*) – *ценой игры*.

Суть метода:

Алгоритм поиска решения матричной антагонистической игры, заданной матрицей выигрышей, имеющей размерность m×n, сводится к алгоритму симплекс-метода решения пары взаимодействующих задач линейного программирования.
Пусть антагонистическая игра задана матрицей выигрышей A, имеющей размерность m×n. Необходимо найти решение игры, т.е. определить оптимальные смешанные стратегии первого и второго игроков: $p^* = (p_1^*, p_2^*, \ldots, p_m^*)$, $q^* = (q_1^*, q_2^*, \ldots, q_n^*)$, где p* и q* - векторы, компоненты которых p_i^* и p_j^* характеризуют вероятности применения чистых стратегий i и j соответственно первым и вторым игроками и соответственно для них выполняются соотношения:

$\sum p_i^* = 1,$

$\sum q_j^* = 1$

Найдём сначала оптимальную стратегию первого игрока p*. Эта стратегия должна обеспечить выигрыш первому игроку не меньше V, при любом поведении второго игрока, и выигрыш, равный V , при его оптимальном поведении, т.е. при стратегии q*. Цена игры V нам пока неизвестна. Без ограничения общности, можно предположить её равной некоторому положительному числу V>0.
Действительно, для того, чтобы выполнялось условие V > 0, достаточно, чтобы все элементы матрицы A были неотрицательными. Этого всегда можно добиться с помощью аффинных преобразований: прибавляя ко всем элементам матрицы A одну и ту же достаточно большую положительную константу M; при этом цена игры увеличится на M, а решение не изменится.
Предположим, что первый игрок A применяет свою оптимальную стратегию p*, а второй игрок B свою чистую стратегию j-ю, тогда средний выигрыш (математическое ожидание) первого игрока A будет равен:

$a_j = a_{ij} p_1^* + \ldots + a_{mj} p_m^*$

Оптимальная стратегия игрока A обладает тем свойством, что при любом поведении игрока B обеспечивает выигрыш первому игроку, не меньший, чем цена игры V; значит, любое из чисел a_j не может быть меньше V. Следовательно, при оптимальной стратегии, должна выполняться следующая система неравенств:

$a_{11} p_1^* + \ldots + a_{mj} p_m^* >= V$

$a_{12} p_1^* + \ldots + a_{mj} p_m^* >= V$

…

$a_{1n} p_1^* + \ldots + a_{mn} p_m^* >= V$

Разделим неравенства на положительную величину V и введём обозначения y_1, …, y_m для новых переменных $p_1/V, \ldots, p_m/V$, $y_1 \geq 0$, $y_2 \geq 0$, .., $y_m \geq 0$. Тогда условия запишутся в виде:

$a_{11}y_1 + \ldots + a_{mj}y_m >= 1$

$a_{12}y_1 + \ldots + a_{mj}y_m >= 1$

…

$a_{1n}y_1 + \ldots + a_{mn}y_m >= 1$

где y1, y2, ..., ym - неотрицательные переменные. В силу неравенств переменные y1, y2 , ..., ym удовлетворяют условию, которое обозначим через F:

$F = \sum y_i = 1/V.$

Поскольку первый игрок свой гарантированный выигрыш V старается сделать максимально возможным, очевидно, при этом правая часть – $1/V \rightarrow \min$. Таким образом, задача решения антагонистической игры для первого игрока свелась к следующей математической задаче: определить неотрицательные значения переменных y1, y2, ..., ym, чтобы они удовлетворяли системе функциональных линейных ограничений в виде неравенств, системе общих ограничений неотрицательности и минимизировали целевую функцию F:

$F = \sum y_i \rightarrow \min$

Это задача линейного программирования и она может быть решена симплекс-методом. Таким образом, решая задачу линейного программирования, можно найти оптимальную стратегию $p^* = (p_1^*, p_2^*, \ldots, p_m^*)$ игрока А. Чтобы найти оптимальную стратегию $q^* = (q_1^*, q_2^*, \ldots, q_n^*)$ игрока В, нужно провести аналогичные действия, с той разницей, что игрок В стремится не максимизировать, а минимизировать проигрыш, а значит, не минимизировать, а максимизировать величину 1/V. Тогда должны выполняться условия:

$a_{11}x_1 + \ldots + a_{mj}x_m <= 1$

$a_{12}x_1 + \ldots + a_{mj}x_m <= 1$

…

$a_{1n}x_1 + \ldots + a_{mn}x_m <= 1$

где $x_1 = q_1/V, \ldots, x_m = q_m/V,$

$x_1 \geq 0, x_2 \geq 0, .., x_n \geq 0.$

Требуется так выбрать переменные $x_1, x_2, .., x_n$, чтобы они удовлетворяли условиям и обращали в максимум линейную функцию цели F':

$F' = \sum x_i = 1/V \rightarrow \max$

Таким образом, задача решения антагонистической игры для второго игрока свелась к следующей математической задаче: определить неотрицательные значения переменных $x_1, x_2, .., x_n$, чтобы они удовлетворяли системе функциональных линейных ограничений в виде неравенств, системе общих ограничений и максимизировали целевую функцию F':

$F' = \sum x_i \rightarrow max$

Это типичная задача линейного программирования и она может быть решена симплекс-методом. Таким образом, решая прямую задачу линейного программирования, мы можем найти оптимальную стратегию $Q^* = (q_1^*, q_2^*, \ldots, q_n^*)$ игрока B.

Алгоритм симплекс-метода заключается в том, что из множества вершин, принадлежащих границе множества решений системы неравенств, выбирается такая вершина, в которой значение целевой функции достигает максимума (минимума). По определенному правилу находится первоначальный опорный план (некоторая вершина области ограничений). Проверяется, является ли план оптимальным. Если да, то задача решена. Если нет, то переходим к другому улучшенному плану - к другой вершине. Значение целевой функции на этом плане (в этой вершине) заведомо лучше, чем в предыдущей.
Алгоритм перехода осуществляется с помощью некоторого вычислительного шага, который удобно записывать в виде таблиц, называемых симплекс-таблицами. Так как вершин конечное число, то за конечное число шагов мы приходим к оптимальному решению.

Этапы:

- **I этап.** Переход к канонической форме задачи линейного программирования путем введения неотрицательных дополнительных балансовых (базисных) переменных. Запись задачи в симплекс-таблицу. Между системой ограничений задачи и симплекс-таблицей взаимно-однозначное соответствие. Строчек в таблице столько, сколько равенств в системе ограничений, а столбцов - столько, сколько свободных переменных. Базисные переменные заполняют первый столбец, свободные - верхнюю строку таблицы. Нижняя строка называется индексной, в ней записываются коэффициенты при переменных в целевой функции. В правом нижнем углу первоначально записывается 0, если в функции нет свободного члена; если есть, то он записывается с противоположным знаком. На этом месте (в правом нижнем углу) будет значение целевой функции, которое при переходе от одной таблицы к другой должно увеличиваться по модулю.

- **II этап.** Проверка опорного плана на оптимальность. Для этого необходимо анализировать строку целевой функции F. Если найдется хотя бы один коэффициент индексной строки меньше нуля, то план не оптимальный, и его необходимо улучшить.

- **III этап.** Улучшение опорного плана. Из отрицательных коэффициентов индексной строки выбирается наибольший по абсолютной величине. Затем элементы столбца свободных членов симплексной таблицы делит на элементы того же знака ведущего столбца. Далее идет построение нового опорного плана. Переход к новому опорному плану осуществляется в результате пересчета симплексной таблицы методом Жордана—Гаусса.

- **IV этап.** Выписывание оптимального решения.

Описание программы:

Функция `linprog` из библиотеки `SciPy` необходима для решения двойственной задачи линейного программирования с помощью симплекс-метода.
Макет функции linprog выглядит так:

```
scipy.optimize.linprog(c, A_ub=None, b_ub=None, A_eq=None, b_eq=None, bounds=None, method='simplex', callback=None, options=None)
```

Она решает следующую задачу линейного программирования с матрицей A:

Минимизировать:

F = c^T * x,

Для системы уравнений

A_ub * x <= b_ub

Рассмотрим подробнее параметры функции:

- `c` Вектор коэффициентов для функции, которую необходимо минимизировать. В нашем случае это единичный вектор.
- `A_ub`
  Матрицы с коэффициентами для системы неравенств.
- `b_ub` Векторы правой части системы неравенств. Для нашей задачи `b_ub` – единичный вектор.

Ход программы:

1)Сначала решаем задачу, описанную в сути метода. Создаем единичные векторы c (со знаком минус) и b, по длине равные первому столбцу и первой строке матрицы соответственно. Для создания данных векторов применяется языковая конструкция Python, называемая генератором. Для нахождения оптимальной стратегии q второго игрока мы подаем на вход функции linprog вектор c, входную матрицу и вектор b. Соответсвенно для поиска стратегии p первого игрока параметры: вектор b, транспонированная матрица, взятая с минусом, вектор c. Для транспонирования матрицы была использована функция transpose() из библиотеки numpy. Далее мы обращаемся к возвращаемым значениям функции linprog через '.x' для получения вектора оптимальной стратегии. Чтобы узнать цену игры - cost, берется отношение единицы к суммарному значению всех элементов вектора p. Функция nash_equilibrium возвращает цену игры, а также векторы p и q домноженные на cost.

Для вывода ответа в виде рациональных дробей используем функцию format_answer со значениями cost, p и q, полученными после вызова функции nash_equilibrium. Применяем метод класса from_float(flt) модуля Fractions, которая представляет точное значение flt, округляя его до десятого знака после запятой. Также используем функцию limit_denominator() для восстановления рационального числа, которое представлено как float.

2)Для визуализации спектров оптимальных стратегий игроков вызывается функция plot(p, q). Внутри функции вызывается команда plt.figure() с параметром figsize, которая добавляет область рисования размера 10 на 10 дюймов. Далее вызываем функцию fig.add_subplot(pos), чтобы добавить оси к фигуре как часть схемы подзаголовка. Аргумент pos представляет собой трехзначное целое число, где первая цифра - это количество строк, второе - количество столбцов, а третий - индекс подзаголовка. Визуализирует спектры команда .stem, отображающая вертикальные линии в каждом месте от базовой линии до длины вектора и помещает там маркер.

3)Для создания пакета в папку необходимо положить несколько модулей (файл с расширением ‘.py’) и создать файл ‘__init__.py’. После создания пакета переходим к сборке, для которой понадобится конфигурационный файл ‘setup.py’. Он предназначен для хранения метаданных о пакете. Создаем дистрибутив исходного кода (tar.gz) или бинарный файл (wheel). Для этого введём - python setup.py sdist bdist_wheel. В результате в папке ‘./dist’ появятся оба формата, при этом имя дистрибутива будет присвоено исходя из имени пакета и его версии.

4)Для написания unit-тестов мы использовали систему тестирования Nose и модуль unittest. В файле nose.py применяется функция assert_equals, сравнивающая между собой два своих параметра. Так как assert_equals не может сравнить два массива, мы сравниваем каждую пару с округлением до пятого знака после запятой, а затем суммируем количество таких пар и проверяем равно ли это значение длине вектора. В файле unit.py мы используем аналогичную функцию .assertAlmostEqual, сравнивающую числа с плавающей запятой, округляя также до 5-го знака.

Необходимые программы (библиотеки):

Библиотеки

- `SciPy` — библиотека с открытым исходным кодом, предназначенная для выполнения научных и инженерных расчётов. Она необходима для функции `linprog` из пакета `scipy.optimize`, который содержит в себе множество алгоритмов оптимизации. Цель функции `linprog` - минимизация линейной объективной функции с линейными ограничениями равенства и неравенства.
- `NumPy` — библиотека с открытым исходным кодом с такими возможностями, как поддержка многомерных массивов (включая матрицы), поддержка высокоуровневых математических функций, предназначенных для работы с многомерными массивами.
- `fractions` – модуль, предоставляющий поддержку рациональных чисел. Нам он необходим для того, чтобы выводить дроби.
  `setuptools` – библиотека с открытым исходным кодом, используемая для создания пакетов.
- `Matplotlib` Для визуализации решений

- `UnitTest, Nose` Для тестирования функций

Программы:

Python 3.6

Anaconda3

Jupyter

Инструкция к запуску

Здесь и далее будем считать, что пакет Anaconda установлен. Переходим в папку Anaconda3\Scripts и вводим в командной строке:

ipython notebook

В результате запустится веб-сервер и среда разработки в браузере.

Выбираем файл hw2.ipynd и запускаем его.

Для запуска unit-тестов в терминале вводим:

nosetests nose.py
python –m unittest unit.py

Участники

Данько Артём
Ковалева Василина
Виль Вадим

1) Написание функции nash_equilibrium: выполняли Данько и Ковалева, выбрали наиболее оптимальное решение.
2) Визуализация спектров оптимальных стратегий: выполнил Данько.
3) Оформление кода в виде пакета: выполнил Данько.
4) Написание unit-тестов для функции nash_equilibrium(a): выполнила Ковалева, изучали систему тестирования Данько и Ковалева.
5) Написание и оформление readme: выполнили Виль, Данько и Ковалева